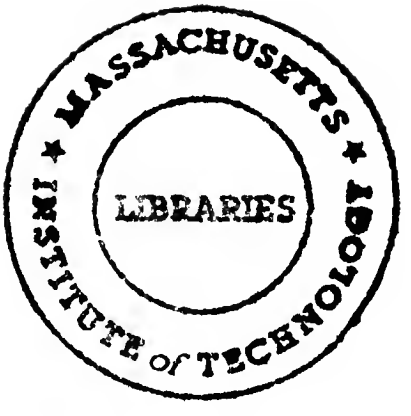

MASSACHUSETTS
INSTITUTE of TECHNOLOGY
LIBRARIES

D28
M414
no. 1863
-87

Dewey

MAR 18 1987

THE ECONOMICS OF SOFTWARE QUALITY ASSURANCE: A SYSTEM DYNAMICS BASED SIMULATION APPROACH

Tarek K. Abdel-Hamid

Stuart E. Madnick

February 1987 #WP 1863-87

THE ECONOMICS OF SOFTWARE QUALITY ASSURANCE: A SYSTEM DYNAMICS BASED SIMULATION APPROACH

Tarek K. Abdel-Hamid

Stuart E. Madnick

February 1987 #WP 1863-87

MAR 1 9

THE ECONOMICS OF SOFTWARE QUALITY ASSURANCE: A SYSTEM DYNAMICS BASED SIMULATION APPROACH

Abstract

The software quality assurance (QA) function has gained, in recent years, the recognition of being a critical factor in the successful development, deployment, and maintenance of software systems. However, because the utilization of QA tools and techniques does tend to add significantly to the cost of developing software, the cost-effectiveness of QA has been a pressing concern to the software quality manager. As of yet, though, this concern has not been adequately addressed in the literature.

Our objective in this paper is to investigate the tradeoffs between the economic benefits and costs of QA efforts in terms of a software project's total development cost. To do this, we developed an integrative System Dynamics model of the software development process. The model is comprehensive in that it integrates the multiple functions of the software development process, including both the management-type functions (e.g., planning, control, and staffing) as well as the software production-type activities (e.g., design and coding). The model also captures the dynamics of error generation as well as the QA activities of error detection and correction.

An important utility of the model is to serve as a laboratory vehicle to conduct controlled experiments on QA policy. Experimental results pertaining to the economics of QA are presented and discussed.

Introduction:

Software quality assurance (QA) is increasingly being perceived as a most critical factor in the successful management of software projects. This is happening because more and more software managers are starting to realize that " QA not only holds the key to a customer's satisfaction, but it also has a direct impact on the cost and the scheduling of a project. Failure to pay attention to QA has often resulted in budget overruns, schedule delays, and failure to meet the needs of the customer" (Chow, 1985).

According to the IEEE standard P730, QA has been defined as "a

planned and systematic pattern of all actions necessary to provide adequate confidence that the software conforms to established technical requirements" (Buckley and Poston, 1984). This definition encompasses two key ideas. First, the definition provides for a comprehensive view of QA, rather than a restrictive one. In other words, the message is that QA is not restricted to say a set of technical methodologies, but rather it includes all the necessary activities (including management techniques, organizational approaches, and administrative procedures) that may contribute to the quality of software during the entire lifecycle of the product. Second, the definition emphasizes the importance of a plan for QA, and of its systematic implementation to achieve an organization's desired objectives of software quality.

In this paper, our focus is on the managerial (not the technical) issues pertaining to the economics of the QA activity. Specifically, we will investigate the tradeoffs between the economic benefits and costs of the QA effort in terms of total project cost. Such considerations obviously lie (implicitly if not explicitly) at the heart of the QA planning process.

The utilization of QA tools and techniques adds significantly to the cost of developing software. For example, man-hours are expended in developing test cases, running test cases, conducting structured walkthroughs, etc. This added cost is

> ... a source of concern to everyone associated with the program, particularly the program manager and the customer ... A (more) pressing concern to the software quality manager is

> how cost efficient are the QA operations during the development cycle. The QA organization, just as all elements of the development process, will and should be subject to detailed and continuing scrutiny regarding the cost of doing business (Knight, 1979).

This "pressing concern" has not, however, been adequately addressed in the literature. That is, as of yet, there are no published studies investigating the cost efficiency of QA operations during the development cycle. Paraphrasing Lawler (1985):

> ... the impact of software quality on development costs has not been explored. Consquently, software costing models do not account for the impact of software quality on development costs.

In stressing the importance of top management's commitment to quality, Riggs (1983) offers an interesting suggestion to QA managers in gaining this commitment. He suggests that the language of top management is finance, or money, but that the language of software production is "things" (e.g., quantity of output, number of labor hours, and number of errors). The challenge to the QA staff is to bridge the gap between these languages, to translate the cost of quality and the opportunity for improved quality into dollars and cents terms. Such reinterpretation of quality is an important step in gaining the involvement and support of top management in the business of improving quality.

In the remaining parts of this paper we propose a new research approach to the study of the software development process in

general and the economics of QA in particular. An overview of our integrative System Dynamics modeling approach of software development is first presented in the section immediately following this introduction. This is then followed by a more detailed discussion of the model's QA section, and the model's experimental results pertaining to the economics of QA.

An Integrative System Dynamics Model of Software Development:

Our research work on the economics of QA is really only one part of a much broader research effort to study the dynamics of the software development process. The overall objective of this ongoing research effort is to develop a scientific base, a theory if you will, of the software project management process. Accomplishments to date include the completion of an extensive series of field interviews of software developers, a compilation of research findings in a comprehensive database, and mostimportantly the development of a System Dynamics computer model of software development project management. The model is currently being used in several research capacities, one of which is to serve as a laboratory vehicle for conducting experimentation in the area of QA, the topic of this paper. Our objective in this section is to provide an overview of the model. A full description of the model's structure, its mathematical formulation, and the validation experiments performed on it are provided in other reports [(Abdel-Hamid, 1984) and (Abdel-Hamid and Madnick, 1987)].

The model is integrative in the sense that it integrates the multiple functions of the software development process, including both the management-type functions (e.g., planning, control, and staffing) as well as the software production-type activities (e.g., design, coding, reviewing, and testing). Figure 1 depicts the model's four major subsystems, namely: (1) The Human Resource Management Subsystem; (2) The Software Production Subsystem; (3) The Controlling Subsystem; and (4) The Planning Subsystem. The figure also illustrates some of the interrelations between the four subsystems.

The Human Resource Management Subsystem captures the hiring, training, assimilation, and transfer of the project's human resource. Such actions are not carried out in vacuum, but, as Figure 1 suggests, they are affected by the other subsystems. For example, the project's hiring rate is a function of the workforce level needed to complete the project on a certain planned completion date. Similarly, what workforce is available has direct bearing on the allocation of manpower among the different software production activities in the Software Production Subsystem.

The four primary software production activities are: development, quality assurance, rework, and testing. The development activity comprises both the design and coding of the software. As the software is developed, it is also reviewed, e.g., using structured walkthroughs, to detect any errors. Errors

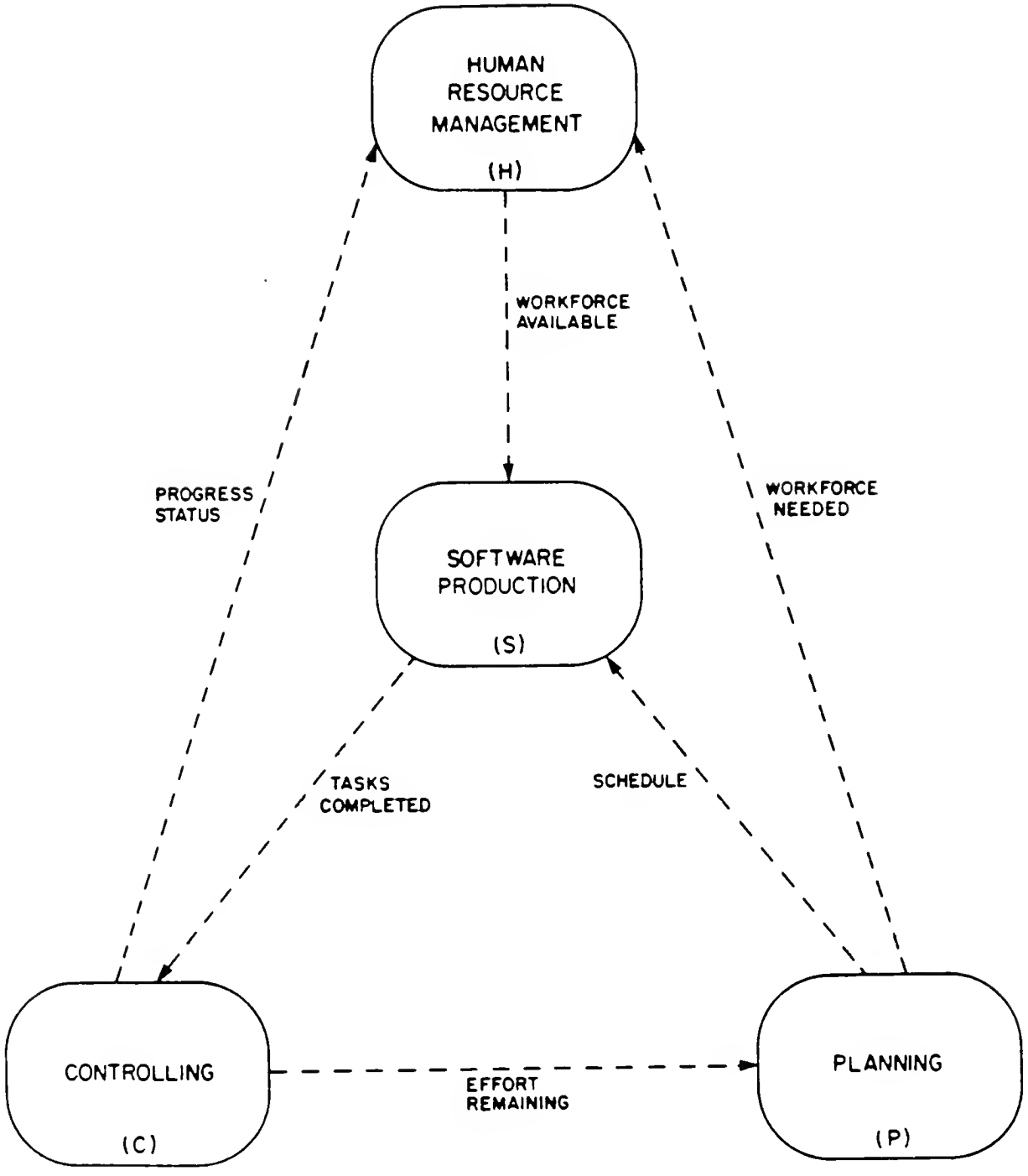

HUMAN
RESOURCE
MANAGEMENT
(H)
WORKFORCE
AVAILABLE
PROGRESS
STATUS
WORKFORCE
NEEDED
SOFTWARE
PRODUCTION
(S)
TASKS
COMPLETED
SCHEDULE
CONTROLLING
(C)
EFFORT
REMAINING
PLANNING
(P)

Figure (1)

detected through such quality assurance activities are then reworked. Not all errors get detected and reworked at this phase, however. Some "escape" detection until the end of development e.g., until the system testing phase.

As progress is made on the software production activities, it is reported. A comparison of where the project is versus where it should be (according to plan) is a control-type activity captured within the Controlling Subsystem. Once an assessment of the project's status is made (using available information), it becomes an important input to the planning function.

In the Planning Subsystem, initial project estimates are made at the initiation of the project, and then these estimates are revised, when necessary, throughout the project's life. For example, to handle a project that is perceived to be behind schedule, plans can be revised to (among other things) hire more people, extend the schedule, or do a little of both.

In addition to being integrative, our modeling approach has a second characteristic feature that distinguishes it from most other work in the software engineering field, namely, the use of the System Dynamics methodology. "System Dynamics is the application of feedback control systems principles and techniques to managerial, organizational, and socioeconomic problems" (Roberts, 1981). As its name implies, System Dynamics is a method of dealing with questions

about the dynamic tendencies of complex systems, that is, the behavioral patterns they generate over time, e.g., whether the system as a whole is stable or unstable, growing, declining, or in equilibrium.

The System Dynamics philosophy is based on several premises [Forrester (1961) and Roberts (1981)]:

1. The behavior (or time history) of an organizational entity is principally caused by its structure. The structure includes, not only the physical aspects, but more importantly the policies and procedures, both tangible and intangible, that dominate decision-making in the organizational entity.
2. Managerial decision-making takes place in a framework that belongs to the general class known as information-feedback systems.
3. Our intuitive judgement is unreliable about how these systems will change with time, even when we have good knowledge of the individual parts of the system.
4. Model experimentation makes it possible to fill the gap where our judgement and knowledge are weakest --- by showing the way in which the known separate system parts can interact to produce unexpected and troublesome overall system results.

Based on these philosophical beliefs, two principal

foundations for operationalizing the System Dynamics technique were established. These are:

1. The use of information-feedback systems to model and understand system structure (premises 1 and 2).

2. The use of computer simulation to understand system behavior (premises 3 and 4).

Consider, as an example, the simple feedback loop of Figure 2. This feedback loop portrays some of the dynamic forces that impact upon the QA activity in the software project environment. The loop shows that the schedule pressures that often arise as a software project falls behind schedule can lead to a higher error generation rate. (More on this later.) As more and more errors are committed, a larger and larger chunk of the available manpower is diverted from development work and devoted instead to error correction and rework duties. As this happens, the project's progress rate drops further, leading to even higher schedule pressures, and another pass around this "vicious cycle." Happily, project managers do have "escape" mechanisms to break loose from the grip of this positive feedback loop. For example, as schedule pressures persist (e.g., after several passes around the loop), project managers could, among other things, add more people, extend the schedule, or do a combination of both.

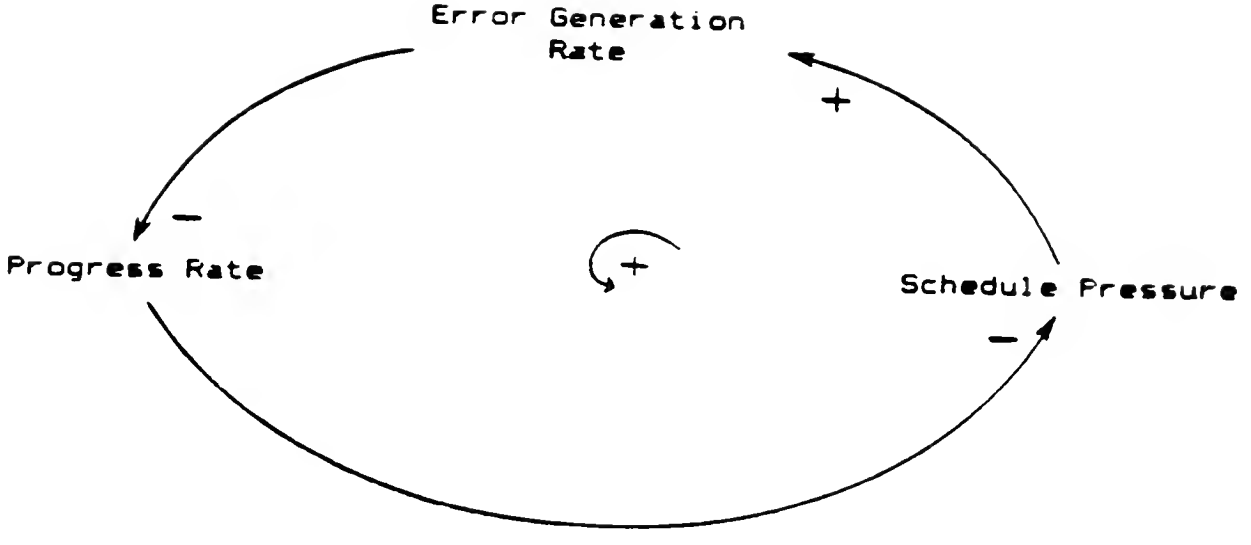


Figure (2)

In another paper (Abdel-Hamid and Madnick, 1986) we discuss in detail the philosophical arguments for the applicability of the feedback systems concepts of System Dynamics to software project management and show how they do provide a powerful lens to view and understand software project behavior.

Error Generation, Detection, and Correction:

Our objective in this section is to present in some detail the model's structures pertaining to the QA activities of error detection and correction, and which lie, as was mentioned above, within the model's "Software Production Subsystem."

Software errors come in many different "flavors." Summarized below are what Nelson (1974) delineated and described as the most prominant software design and coding errors:

- Misinterpretation of specifications
- Errors in developing the logic to solve the problem
- Algorithm approximations that may provide insufficient accuracy or erroneous results for certain input variables
- Data structure defects either in the data structure design specifications or in the implementation of the specifications
- Singular or critical input values to a formula that may yield an unexpected result not accounted for in the program code
- Misinterpretation of language constructions by the programmer.

In any System Dynamics model it is quite feasible, and in fact straight forward from a technical point of view, to disaggregate a variable such as the error variable into more than one error type. However, it is not always necessary or useful:

> There are two (and only two) considerations for reformulating a level (variable) as a sequence of two or more levels: policy analysis and model behavior. First, is the disaggregation required in order for the model to be able to address particular policy issues? ...
> The second reason for disaggregating a level (variable) involves the dynamics of the system. Does the disaggregation of a level into two or more levels has the potential to change significantly the behavior of the model? ...
> The final arbiter should be model-based policy analysis. If the change in behavior has the potential to alter policy conclusions, then the disaggregation is essential (Richardson and Pugh, 1981).

Since our model's policy focus is on the managerial-type policies of software development, as opposed to the technological issues of software reliability, an explicit disaggregation of errors into more than one type is, on the basis of the policy analysis criterion, clearly unnecessary. On the other hand, there are significant behavioral differences among error types that must be accounted for. For example, findings in the software engineering literature indicate that, at different points in the lifecycle, errors are generated at different rates, e.g., design errors are generated at a higher rate than are coding errors (Martin, 1982). Such a factor is obviously of dynamic significance, e.g., it could have a direct bearing on the allocation of the manpower resource

for error correction activities, which in turn would affect the software development rate and hence the project's progress.

Such differences are implicilty captured in the model. That is, while errors are formulated as a single variable (ERRORS), the generation, detection, and correction characteristics of ERRORS do vary throughout the development lifecycle. For example, "ERRORS" are generated at a higher rate in the earlier portions of the lifecycle (as design errors do) and they are, on the average, harder to detect and correct (as design errors are).

Figure 3 depicts the model's structure for the generation, detection, and correction of errors. The figure demonstrates as well the interrelationships between this part of the model and two other sectors, namely, the "software development" and "system testing" sectors. Also note that these three sectors together constitute the "Software Production Subsystem" of Figure 1.

System Dynamics Modeling Conventions:

The schematic conventions used in Figure 3 are the standard conventions used in System Dynamics models. From a System Dynamics perspective all systems can be represented in terms of "level," "rate," and "auxiliary" variables.

A level is an accumulation, or an integration, over time of flows or changes that come into and go out of the level. The term "level" is intended to invoke the image of the level of a liquid

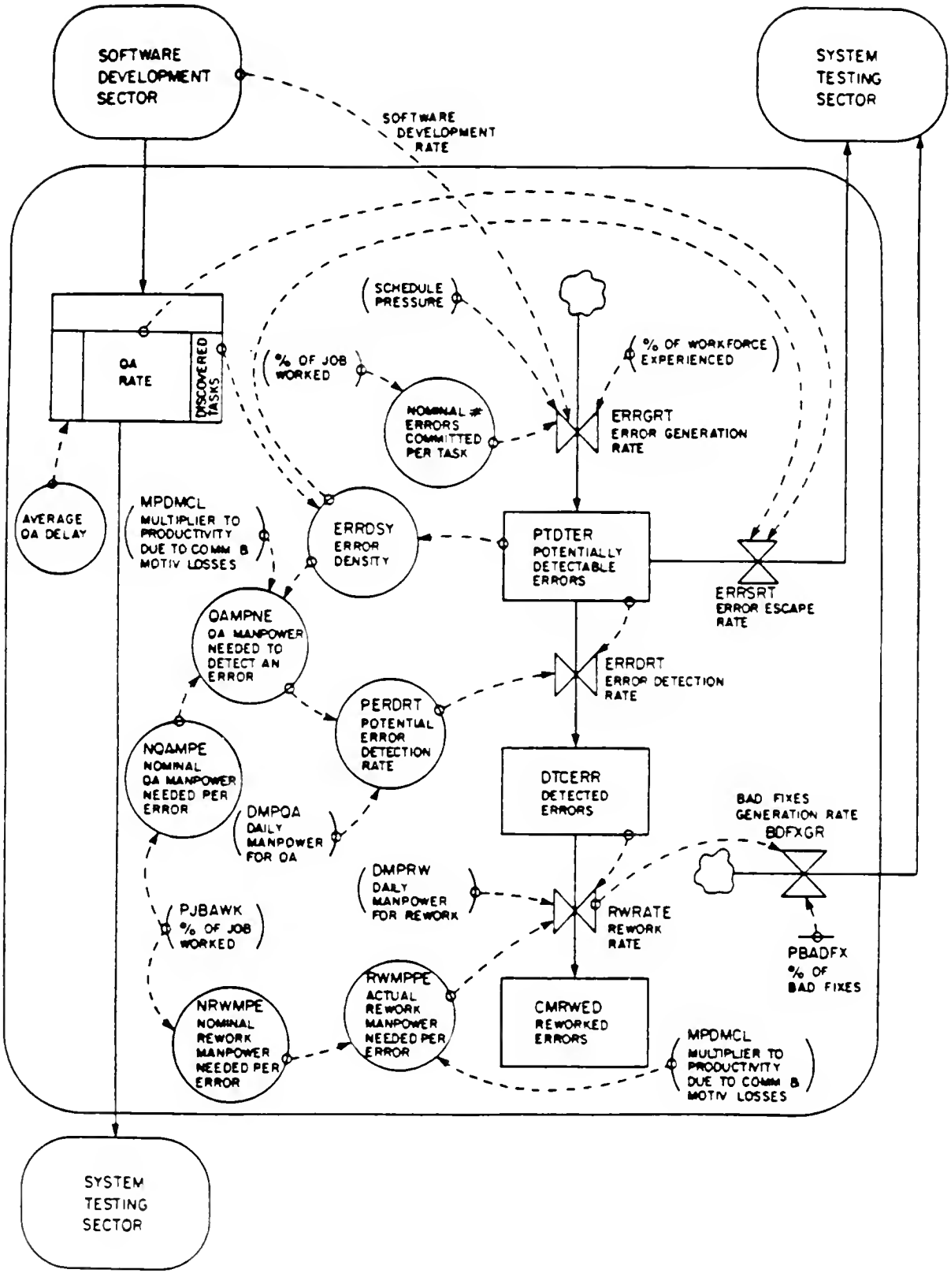
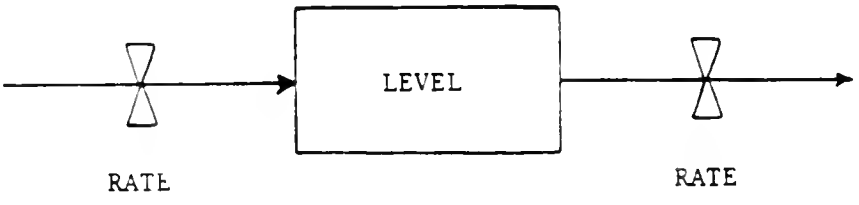

SOFTWARE DEVELOPMENT SECTOR
SOFTWARE DEVELOPMENT RATE
SYSTEM TESTING SECTOR
QA RATE
DISCOVERED TASKS
SCHEDULE PRESSURE
% OF JOB WORKED
NOMINAL # ERRORS COMMITTED PER TASK
% OF WORKFORCE EXPERIENCED
ERRGRT ERROR GENERATION RATE
AVERAGE QA DELAY
MPDMCL MULTIPLIER TO PRODUCTIVITY DUE TO COMM & MOTIV LOSSES
ERRDSY ERROR DENSITY
PTDTER POTENTIALLY DETECTABLE ERRORS
ERRSRT ERROR ESCAPE RATE
QAMPNE QA MANPOWER NEEDED TO DETECT AN ERROR
ERRDRT ERROR DETECTION RATE
PERDRT POTENTIAL ERROR DETECTION RATE
NQAMPE NOMINAL QA MANPOWER NEEDED PER ERROR
DTCERR DETECTED ERRORS
BAD FIXES GENERATION RATE
BDFXGR
DMPQA DAILY MANPOWER FOR QA
DMPRW DAILY MANPOWER FOR REWORK
RWRATE REWORK RATE
PJBAWK % OF JOB WORKED
PBADFX % OF BAD FIXES
RWMPPE ACTUAL REWORK MANPOWER NEEDED PER ERROR
NRWMPE NOMINAL REWORK MANPOWER NEEDED PER ERROR
CMRWED REWORKED ERRORS
MPDMCL MULTIPLIER TO PRODUCTIVITY DUE TO COMM & MOTIV LOSSES
SYSTEM TESTING SECTOR

Figure (3)

accumulating in a container. The flows increasing and decreasing a level are called rates. Thus, "DETECTED ERRORS" is a level of errors that is increased by the "ERROR DETECTION RATE" and decreased by the error "REWORK RATE."

Rates and levels are represented as stylized valves and tubs, as shown below, further emphasizing the analogy between accumulation processes and the flow of a liquid.



The flows that are controlled by the rates are usually diagrammed differently, depending on the type of quantity involved. We will use the two types of arrow designators shown below:

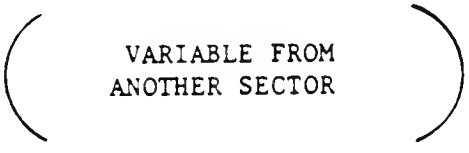
INFORMATION FLOWS — — — — — →

OTHER FLOWS (e.g., PEOPLE) ——————→

All tangible variables are either levels or rates i.e., they are either accumulations of previous flows or are presently flowing. Auxiliary variables, on the other hand, are information-type variables in the system, and capture things like concepts (e.g., the concept of "ERROR DENSITY") and policies (e.g., the policy for allocating "DAILY MANPOWER FOR REWORK"). Auxiliary

variables are represented by a circular symbol.

Finally, variables that are defined in other sectors of the model are either shown as arcs emanating from their respective sectors (e.g. the "SOFTWARE DEVELOPMENT RATE" variable from the SOFTWARE DEVELOPMENT SECTOR) or are represented by enclosing the variable name in parenthess as shown below.



Error Generation Factors:

Returning back to Figure 3, consider the variable "ERROR GENERATION RATE." There are two sets of factors that affect the error generation rate in a software project. The first set includes: organizational factors [e.g., an organization's use of structured techniques (Alberts, 1976), the overall quality of the staff (Belford et al, 1977), etc.] and project-specific-type factors [e.g., project complexity, system size, programming

language, etc.]. Notice that even though such factors can differ from organization to organization and from one project to another, they do, however, tend to remain invariant during the life of any particular project. From our modeling viewpoint, this observation is quite significant. It means that, in modeling the behavior of a single software development project, most of the above variables would remain constant and their effect could, therefore, be captured by a single nominal variable, namely, the "NOMINAL NUMBER OF ERRORS COMMITTED PER TASK." (Task is a measure of a unit of work, e.g., a software module, a block of 30 lines of code, etc.) Such a nominal variable would require modification only when modeling different organizational settings or different projects, but not when experimenting with policies (such as QA policy) for a particular software project (which is the scenario in this paper).

Given a certain "NOMINAL NUMBER OF ERRORS COMMITTED PER TASK," the nominal error generation rate would then simply be the product of the "SOFTWARE DEVELOPMENT RATE," i.e., how much tasks are developed per unit of time, and the "NOMINAL NUMBER OF ERRORS COMMITTED PER TASK." In order to capture the generation of different error types, the "NOMINAL NUMBER OF ERRORS COMMITTED PER TASK" is not formulated as a constant number, but rather as a dynamic variable that changes over the project's life. (Project phase is captured in the modelthrough the continuous variable "% OF JOB WORKED.") The formulation of the "NOMINAL NUMBER OF ERRORS COMMITTED PER TASK" therefore serves two purposes. First, its shape

over the project's life reflects the relative generation rates of different error types throughout the life of a project. The second purpose of the formulation (its absolute value) reflects the different error generation characteristics of different project situations (i.e., the software product's characteristics as well as those of the organization in which it is developed). This, obviously, would generally change when modeling different project settings. (The parameter profile that characterizes the software project (EXAMPLE) used in our experimentation, is presented in the Appendix.)

In addition to the organizational and project-specific factors discussed above, there is a second set of factors which affect error generation rates and which, unlike the previous set, do not remain invariant during the life of a project, but rather do play a dynamic role during software development. These include the workforce-mix and schedule pressures.

The workforce level in the model is disaggregated into two types of employees, newly hired and experienced. Newly hired project members (from outside the organization or from within it) typically pass through a project orientation phase during which they are less than fully productive. The orientation process brings them up to speed through training that covers both the social as well as the technical environments of the project (Couger and Zawacki, 1980). While not yet fully trained (during this

orientation period) newly hired employees are not only, on the average, less productive, they also tend to be more error-prone than their experienced counterparts [(Endres, 1975) and (Myers, 1976)]. The effect of this factor is captured by the "MULTIPLIER TO ERROR GENERATION DUE TO WORKFORCE MIX." (In Figure 3, this multiplier is represented by the arc from "% OF WORKFORCE EXPERIENCED" to "ERROR GENERATION RATE.") It is a variable that varies, by as much as 100%, as a function of the "% OF WORKFORCE EXPERIENCED." That is, it is assumed in the model [based on the research findings reported in (Abdel-Hamid, 1984) and (Abdel-Hamid and Madnick, 1987)] that a newly hired employee is, on the average, twice as error-prone as an experienced employee would be.

The second factor that can drive error generation up is schedule pressure [(Mills, 1983), (Putnam and Fitzsimmons, 1979), and (Radice, 1982)]. Paraphrasing DeMarco (1982):

> People under time pressure don't work better, they just work faster ...
> In the struggle to deliver any software at all, the first casuality has been consideration of the quality of the software delivered.

Two explanations have been proposed in the literature for why schedule pressures cause more errors to be generated. First, Schneiderman (1980) suggests that the schedule pressures increase the "anxiety levels" of programmers. A high anxiety level, then

> ... interfaces (with performance), probably by reducing the size of the short-term memory available. When programmers

> become more anxious as deadlines approach, they (therefore) tend to make even more errors ...

Another explanation was provided by Thibodeau and Dodson (1980). They suggest that schedule pressures often result in the overlapping of activities that would have been accomplished better sequentially, and that this can significantly increase the chance of errors. For example,

> When coding has begun before the completion of design, the designers are required to communicate their results to the programmers in a raw, unqualified state, hence significantly increasing the chance of design errors ...
> This is not to suggest that systems cannot be developed with overlapping activities. Many systems have distinct parts that can be coded before the entire design is completed ... We are concerned here with the situation where the press of the development schedule or the slippage of preceding activities results in overlapping activities that would have been accomplished better sequentially (Thibodeau and Dodson, 1980).

The effect of schedule pressure on error generation is captured in the model through the "MULTIPLIER TO SCHEDULE PRESSURE." (In Figure 3, this multiplier is represented by the arc from "SCHEDULE PRESSURE" to "ERROR GENERATION RATE.") Under nominal conditions where there are no schedule pressures (e.g., when the project is <u>perceived</u> to be on schedule), the multiplier assumes a neutral value. As schedule pressures increase in a project (e.g., as deadlines are approached), the multiplier increases exponentially leading to higher error generation rates, which our field studies revealed could (under severe schedule pressures) be as much as 50% higher than nominal.

Thus, as software tasks are developed, errors are committed within those tasks. Errors within a developed task remain as "POTENTIALLY DETECTABLE ERRORS" until the task is reviewed and tested, at which point some of the errors do get detected, and these are then reworked. Usually, though, not all errors will be detected, some will "escape" and pass undetected into the subsequent phases of software development, where they might then be caught, albeit at a relatively much higher cost.

Error Detection Factors:

The detection of errors is the objective of the software quality assurance (QA) activities. The "QA RATE" shown in Figure 3, has a rather non-characteristic mathematical formulation (with its special schematic representation), namely, that of an exponential delay. [A delay is essentially a conversion process that accepts a given inflow rate and delivers a resulting flow rate at the output. The outflow may differ instant by instant from the inflow rate under dynamic circumstances where the rates are changing in value. This necessarily implies that the delay contains a variable amount of the quantity in transit, e.g., software to be QA'ed. For a more detailed discussion of the mathematical formulation and behavior characteristics of exponential delays in System Dynamics models see Forrester (1961).]

The "characteristic" way to formulate a rate of accomplishing something such as the rate of developing software or correcting

errors, is to formulate it as a product of the effort allocated to the activity and the productivity at which this effort is utilized. However, what our field studies uncovered (and what the exponential delay formulation captures) is that the QA rate tends to be independent of the allocated QA effort and its productivity! What we found happening in the organizations we studied is this: QA effort is planned and allocated, usually in the form of a fixed schedule of periodic group-type functions. For example, a two-hour walkthrough for project members would be scheduled once a week. During these periodic "QA windows," all tasks developed since the previous one are supposed to be processed. And what we were surprised to find was that irrespective of how many tasks needed to be processed within the particular "QA window," they almost always were "processed." No backlogs, therefore, develop in the QA pipeline. Even when QA activities are relaxed or suspended temporarily because of schedule pressure, no backlogs develop. For example, when walkthroughs are suspended for a while on a project, the requirement to review the affected tasks is bypassed, not postponed. [This behavior was also reported by others in the literature e.g., (Hart, 1982) and (Mitchell, 1980).]

We can propose an explanation for how and why this happens. Since the objective of the QA activity is to detect invisible errors (invisible that is until they are detected), it becomes almost impossible to tell whether the QA job was completely done (i.e., that all these invisible errors were in fact detected). By

the same token, it is as difficult to tell that the job has not been completely done (except much later in the lifecycle). Under such circumstances it becomes quite easy to rationalize both to oneself and to management that the QA job that was "convenient" to do, was not insufficient. Furthermore, the QA effort that is convenient to expend (i.e., in terms of available time and effort), is usually what is actually expended and not more (e.g., even if more is called for due to a larger than expected workload of developed tasks) because there seems to be no significant incentives to do otherwise. Firstly, at a psychological level, there are actually dis-incentives for working harder at QA, since it only "exposes" more of one's mistakes (Weinberg, 1971). And secondly, at the organizational level, there are seldom any real reward mechanisms in place to really promote quality or quality-related activities (Cooper and Fisher, 1979).

The formulation of the "QA RATE" as an exponential delay provides, we feel, a good approximation of this "Parkinsonian execution" of the QA activity. That is, software tasks that are developed will always be QA'ed (or, more accurately, considered QA'ed) after a certain delay, which is independent of the actual QA effort allocated.

Therefore, the rate at which tasks are considered QA'ed can, under such currently adopted QA practices, proceed independently of the actual QA effort allocated. However, the effectiveness of QA

will, obviously, depend on that effort. That is, the amount of errors detected will be a function of how much QA effort is allocated for error detection.

What are the determinants of the "QA MANPOWER NEEDED TO DETECT AN ERROR?" As was the case with the "ERROR GENERATION RATE," there are organizational-type factors such as the overall quality of the staff, as well as project-specific-type factors such as project complexity and programming language. And as was explained before, all such factors do tend to remain invariant during the life of any particular software project and are, therefore, captured in the model as a single nominal variable, namely, the "NOMINAL QA MANPOWER NEEDED PER ERROR." Because different error types do differ in how costly they are to detect, this nominal variable is not a constant number, but rather it is a dynamic variable that assumes different values as the project progresses through its lifecycle. (See the Appendix.) Specifically, design-type errors are not only generated at a higher rate, as was discussed above, but they are also more costly to detect than coding-type errors [(Alberts, 1976), (Boehm et al, 1975), and (Myers, 1976)].

The actual QA manpower needed to detect an error, in addition to being a function of error-type, must also depend on the efficiency of how people work. A full-time employee's 8-hour work day does not typically translate into a fully productive 8-hour contribution to the project. Man-hours are lost on communication

and other non-project activities (e.g., personal business, coffee breaks, etc.). These types of losses are captured in the model's "MULTIPLIER TO PRODUCTIVITY DUE TO COMMUNICATION AND MOTIVATION LOSSES," which simply represents the average productive fraction of a man-day. In other words, if the communication and motivation losses amount to a 4 man-hour loss per day (for the average project member) , which would be half of the nominal 8-hour value, then the value of the multiplier would be 0.5. Under such circumstances, the actual QA manpower needed to detect an error becomes twice what is nominaly needed. That is, if a design error requires, under nominal conditions (i.e., under conditions of no losses), 0.4 man-days of effort to be detected, it would actually require (in the above case) 0.4 / 0.5 = 0.8 man-days.

Finally, evidence suggests that "In any sizable program, it is impossible to remove all errors" (Shooman, 1983). Thus, even when generous effort allocations are made to QA, it would still be unlikely that all errors will be detected (Boehm, 1981). One reason, for example, is that some errors manifest themselves, and can be exhibited only after system integration (Shooman, 1983). At any point in time one could, therefore, view the set of "POTENTIALLY DETECTABLE ERRORS" as constituting a hierarchy of errors, in which some are more subtle, and therefore more expensive to detect than others. Empirical results reported by Basili and Weiss (1982) suggest that the distribution is pyramid like, with the majority of errors requiring approximately a few hours to

detect, a few errors requiring approximately a day to detect, and still fewer errors requiring more than a day to detect. (Notice that the results show that these few subtle errors are an order of magnitude more expensive to detect.)

We assume in the model that as QA activities are performed, the more obvious errors will be detected first. As these are detected, it then becomes more and more expensive to uncover the remaining more subtle (although less pervasive) errors. This is achieved in the model through the formulation of the "MULTIPLIER TO DETECTION-EFFORT DUE TO ERROR DENSITY." (In Figure 3, this multiplier is captured by the arc from "ERROR DENSITY" to "QA MANPOWER NEEDED TO DETECT AN ERROR.") At moderate-to-large error densities, the multiplier assumes a neutral value. But as the "obvious" errors are detected, the multiplier increases in an exponential fashion, such that the remaining few subtle errors are an order of magnitude more expensive to detect.

To recapitulate, the "QA MANPOWER NEEDED TO DETECT AN ERROR" is a function of error-type, work efficiency, and error density. As the value of this needed effort increases, e.g., due to a decrease in error density, the number of errors that can be detected, at some level of QA effort, decreases. At any point in time, the "POTENTIAL ERROR DETECTION RATE" (determined simply by dividing the QA effort allocated by the value of the "QA MANPOWER NEEDED TO DETECT AN ERROR"), represents the maximum possible number of errors

that could be detected. Because manpower allocations to QA are often modest, this maximum value is seldom large enough to ensure the detection of all errors generated. And even when effort is allocated generously to QA, a few subtle errors will just be too prohibitaviley expensive to detect. As a result, some errors inevitably will "escape" and pass undetected into subsequent phases of software development, as is shown in Figure 3.

Error Generation Factors:

Those errors that do get detected through QA are then reworked. The rework rate is a function of how much effort is allocated to rework activities, and the rework manpower needed per error. For example, if the project members commit 10 man-days per week to rework detected errors, and the "ACTUAL REWORK MANPOWER NEEDED PER ERROR" is, on the average, 1 man-day, then errors will be reworked at the rate of 10 per week.

The "ACTUAL REWORK MANPOWER NEEDED PER ERROR" has two components. The first is the "NOMINAL REWORK MANPOWER NEEDED PER ERROR." As in the case of error detection, this nominal component is a function of error-type i.e., design versus coding errors. Design-type errors, in addition both to being generated at a higher rate and to being more costly to detect, are also more costly to rework [(Alberts, 1976), (Boehm et al, 1975), and (Myers, 1976)]. (See the Appendix.)

The actual rework man-power needed to correct an error, in addition to being a function of error-type, must also depend on the efficiency of how people work. That is, we need to account for the communication and motivation losses incurred. For example, if the "MULTIPLIER TO PRODUCTIVITY DUE TO COMMUNICATION AND MOTIVATION LOSSES," which represents the average productive fraction of a man-day, is 0.5, then the actual rework manpower needed to correct an error becomes twice what is nominally needed.

To recapitulate, as errors are detected through the QA activities, they are reworked. The rate at which errors are reworked is a function of the manpower committed to the rework activity and the rework effort needed per error. The "ACTUAL REWORK MANPOWER NEEDED PER ERROR" is, in turn, a function of two things, error-type and work efficiency.

The reworking of software errors is not, itself, an errorless activity:

> Human tendency is to consider the 'fix,' or correction, to a problem to be error-free itself. Unfortunately, this is all too frequently untrue in the case of fixes to errors found by inspections and by testing (Fagan, 1976).

The problem of bad-fixes is widely documented in the literature [e.g., (Endres, 1975), (Fagan, 1976), (Jones, 1978), (Myers, 1976), and (Shooman, 1983)]. Shooman and Natarajan (1977) suggested some of the ways in which bad-fixes may be generated:

1. The correction is based upon faulty analysis, thus complete bug removal is not accomplished.
2. The corrections of a bug may work locally only (i.e., the global aspects of the error still remain).
3. The correction is accomplished, however, it is accomplished by the creation of a new error.

Thus, as detected errors are reworked, some fraction of the corrections will be bad-fixes. The detection and correction of such bad-fixes, together with that of errors that escape QA detection during the project's development phases, are activities that are captured in other sections of the model (the "System Testing Sector").

For further details on the model's mathematical formulations, the parameter values, and the research findings supporting both, the reader is refered to Abdel-Hamid (1984) and Abdel-Hamid and Madnick (1987).

Model Experimentation:

The above discussion of the model's error generation, detection, and correction structures (which is only one of the three sectors of the "Software Production Subsystem," which in turn is only one of four major subsystems in the model) should

demonstrate the high complexity of such an integrative System Dynamics model. The behavior of such dynamic models is complex beyond the capacity of human intuition (Roberts, 1981). To handle this high complexity, system dynamicists rely on a powerful set of computer simulation tools.

Simulation's particular advantage is its greater fidelity in modeling processes, making possible both more complex models and models of more complex systems. It also allows for less costly and less time-consuming experimentation. Because "... in software engineering it is remarkably easy to propose hypotheses and remarkably difficult to test them" (Weiss, 1979), several authors have argued for the desirability of having such a laboratory tool for testing ideas and hypotheses (Thayer, 1979). Paraphrasing Forrester (1961):

> The effects of different assumptions and environmental factors can be tested. In the model system, unlike the real systems, the effect of changing one factor can be observed while all other factors are held unchanged. Such experimentation will yield new insights into the characteristics of the system that the model represents. By using a model of a complex system, more can be learned about internal interactions than would ever be possible through manipulation of the real system. Internally, the model provides complete control of the system's organizational structure, its policies, and its sensitivities to various events.

In the remaining part of this section we will utilize the model to conduct a series of simulation experiments to investigate the tradeoffs between the economic costs and benefits of QA. The

prototype software project used in the simulation experiments, called project EXAMPLE, is 64,000 delivered source instructions in size (with the QA parameter profile provided in the Appendix).

An important relationship to investigate, obviously, is the one between the QA effort expended in a software project and the percentage of errors detected during development. Several studies have established the significant cost savings gained by the early detection and correction of errors. For example, in a study by Shooman reported in McClure (1981), it was determined that detecting and correcting a design error during the design phase (i.e., through the QA activities) is one-tenth the effort that would be needed to detect and correct it later during the system testing phase because of the additional inventory of specifications, code, user and maintenance manuals, etc., that would require correction in the later case. A primary goal of QA, therefore, is "that errors be detected and corrected as early as possible and only a minimal amount of problems be allowed to slip from one phase of the development to the next" (Tsui and Priven, 1976).

The relationship between QA effort expended and the percentage of errors detected obtained from model experimentation is shown in Figure 4. The significant feature of this result is the "diminishing returns" of QA exhibited as QA expenditures extend beyond 20-30% of development effort. This type of behavior has been

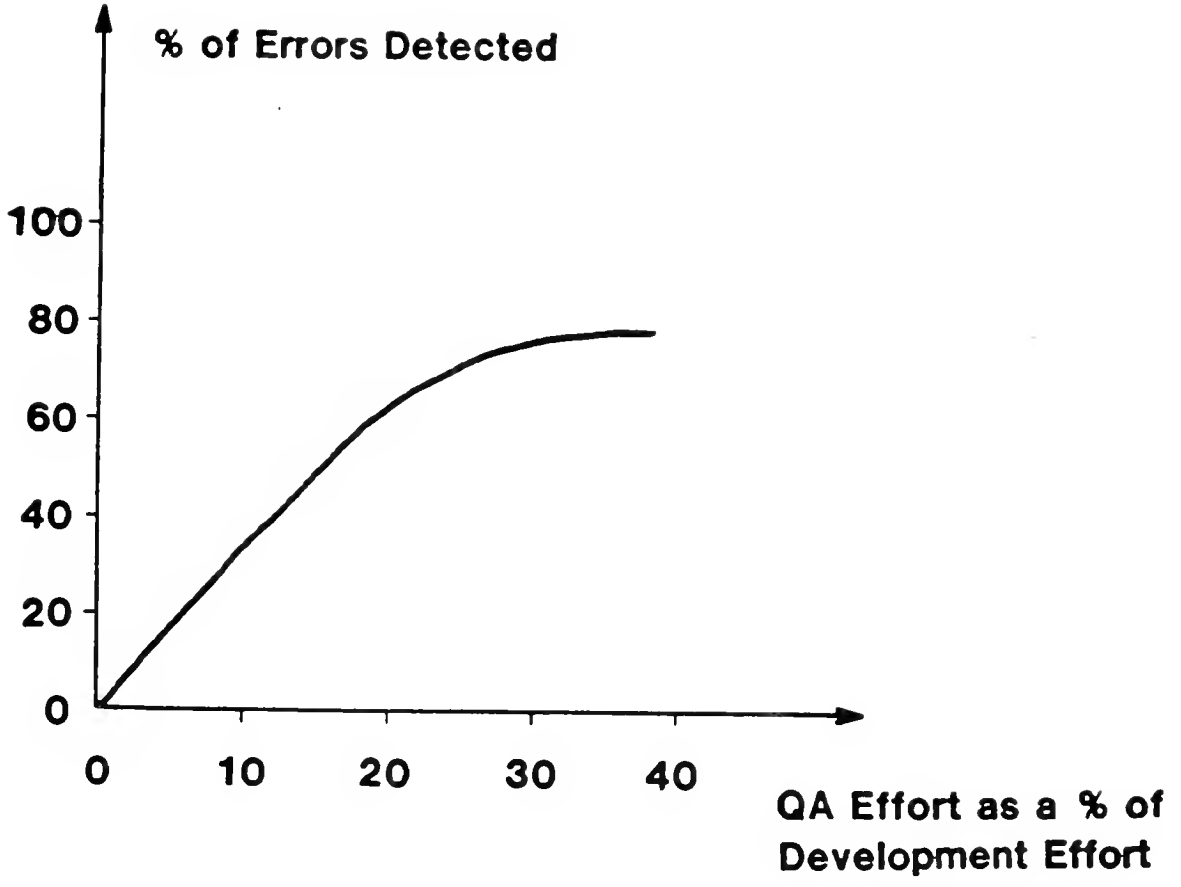

% of Errors Detected
100
80
60
40
20
0
0
10
20
30
40
QA Effort as a % of
Development Effort

Figure (4)

observed by others in the literature [e.g., (Shooman, 1983) and (Boehm, 1981)].

What the results of Figure 4 suggest is that the savings in the cost of processing errors that result from the application of QA, flattens out as QA expenditures extend beyond 20-30% of development effort. This is evident from the cost patterns of Figure 5. As can be seen, the combined costs of rework (i.e., correcting errors during development) and system testing (at the end of development) flatten out as QA expenditures exceed 20%. On the other hand, notice that increasing QA as a percentage of the development effort results in an exponential (not a linear) increase in QA's absolute cost (in man-days). The reason this happens is that as a larger fraction of the development effort is allocated to QA, the development effort itself increases. To see why, consider the sequence of steps typically followed in planning a project's various activities. First, total man-days is estimated. Based on this global value, the project's schedule is calculated. The two estimates are then used to determine the project's average staff size. Allocations are then made to the project's various lifecycle activities (including the QA activity). Notice that effort distribution decisions typically come after, not before, the project's schedule is made (Boehm, 1981). Thus if two different project managers were to run the same software project (e.g., project EXAMPLE), and if the only thing that differentiates them is their policies on the percentage of the development effort to

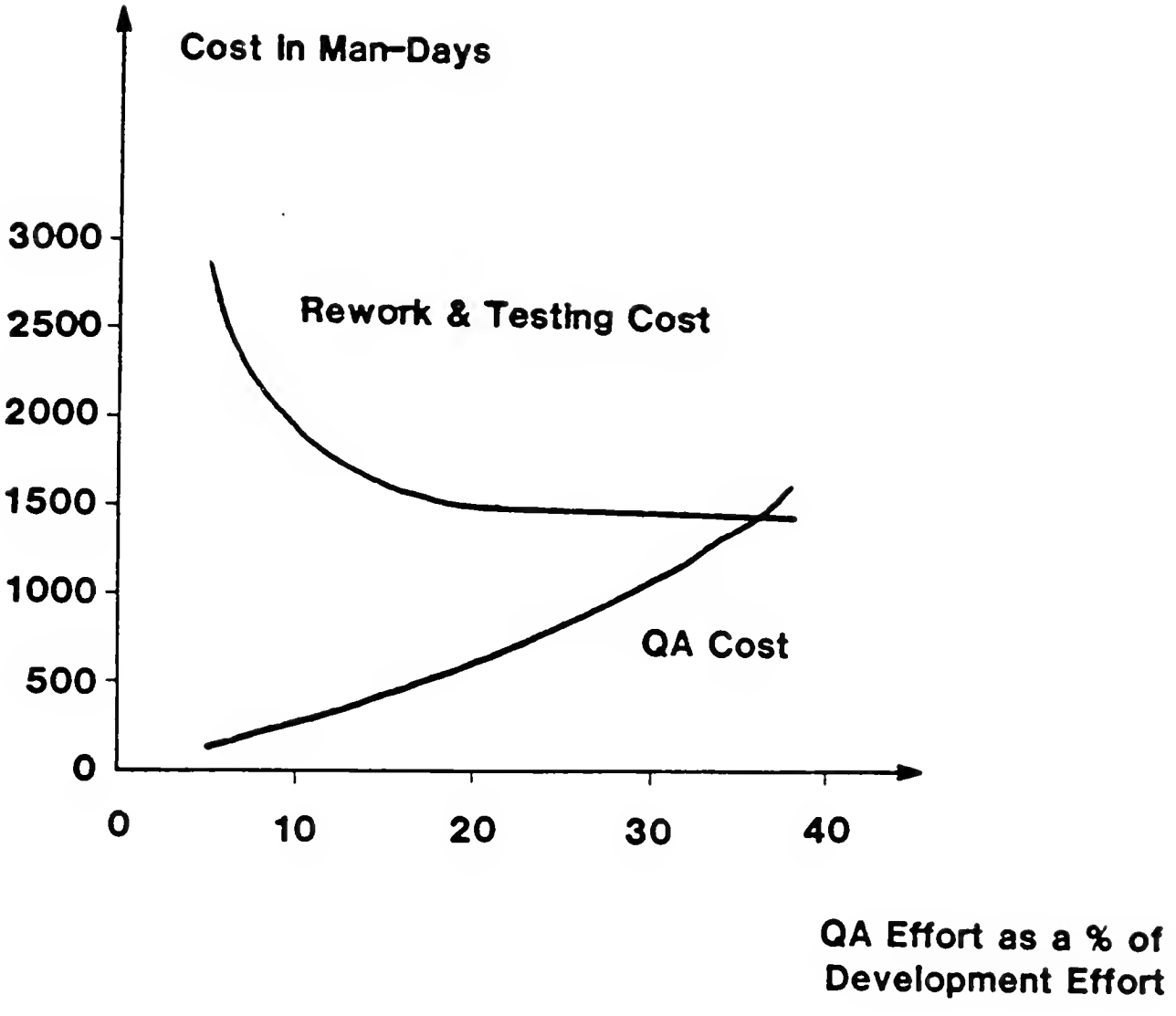

Cost in Man-Days
3000
2500
2000
1500
1000
500
0
Rework & Testing Cost
QA Cost
0
10
20
30
40
QA Effort as a % of
Development Effort

Figure (5)

allocate to QA, both managers would still initiate their respective projects with the same global estimates, i.e., the total man-days and the overall project schedule. It is exactly this type of scenario that our simulation experiment is designed to capture. Thus, in the different runs of the model, the project's total man-day estimate as well as the scheduled completion date remain the same. But since increases in the QA-effort allocation mean that less manpower effort will be available for development work (e.g., design and coding), a larger team will be required to meet the given schedule. A larger team means larger training and communication overheads, and hence the larger development cost.

The final, and perhaps more interesting, issue we addressed in our QA experiments concerns the "optimal" QA-effort expenditure. For our prototype project EXAMPLE, the answer is shown in Figure 6, which plots EXAMPLE's total cost (in man-days) against QA-effort expenditures (defined in terms of percentage of development man-days). As can be seen, the optimal QA effort expenditure for this project is 16% of total development effort (in man-days).

Two important conclusions can be drawn from Figure 6. The first, more generalizable conclusion, is that QA policy does have a significant impact on total project cost. As can be seen from the figure, project EXAMPLE's cost ranges from a low of 3,770 man-days, to values in the range of 5,000 man-days i.e., values that are 33% higher. At low values of QA expenditures, this increase in cost

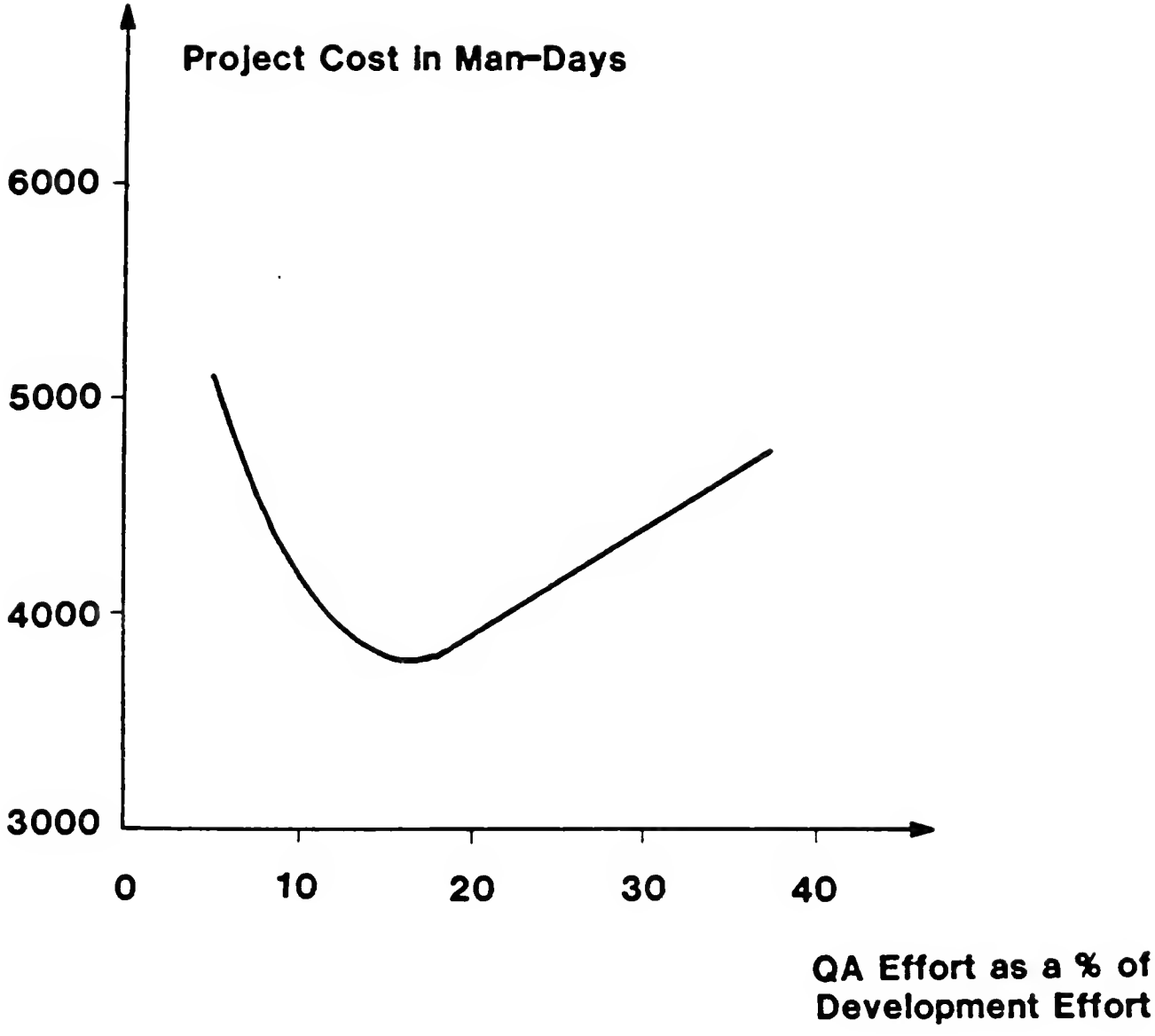

Project Cost in Man-Days
6000
5000
4000
3000
0
10
20
30
40
QA Effort as a % of
Development Effort

Figure (6)

results from the large cost of the testing phase. On the other hand, at high values of QA expenditures, the excessive QA expenditures (and which at high values are less productive because of the diminishing returns phenomenon) are themselves the culprit.

The second result is, of course, deriving the optimal QA expenditure level of 16%. What, in our opinion, is really significant about this result is not its particular value, since this cannot be generalized beyond this experiment's EXAMPLE software project, but rather the process of deriving it, namely, this paper's integrative System Dynamics simulation approach. Beyond controlled experimentation (which would be too costly and time consuming to be practically feasible), as far as we know, this model provides the first capability to quantitatively analyze the costs/benefits of QA policy for software production. And this, it is encouraging to note, is generalizable, in the sense that one can customize models for different software development environments to derive environment-specific optimality conditions.

Summary:

The QA function has, in recent years, gained the recognition of being a critical factor in the successful development of software systems. However, because the utilization of QA tools and techniques does tend to add significantly to the cost of developing software, the cost-effectiveness of QA has been a pressing concern to the software quality manager. As of yet, though, this concern

has not been adequately addressed in the literature.

Our objective in this paper was to investigate the tradeoffs between the economic benefits and costs of QA efforts in terms of a software project's total development cost. To do this, we developed an integrative System Dynamics model of the software devlopment process. The model is comprehensive in that it integrates the multiple functions of the software development process, including both the management-type functions (e.g., planning, control, and staffing) as well as the software production-type activities (e.g., design and coding). The model also captures the dynamics of error generation as well as the QA activities of error detection and correction.

An important utility of the model is to serve as a laboratory vehicle to conduct controlled experiments on QA policy. Experimental results pertaining to the economics of QA reported in this paper show how QA policy impacts total project costs. For the specific example analyzed, the optimal QA effort was shown to be 16% of the total development effort. Although that particular value only applies to the example project, the analysis process using system dynamics is generalizable.

APPENDIX

QA Parameter Profile for Software Project EXAMPLE

Table equations represent a simple way of expressing relationships, particularly nonlinear relations, between variables in a System Dynamics model. Table equations have the following format:

```
Y-variable = TABLE ( Table-name , X-variable , L , H , I )
```

The above equation indicates a functional relationship between an independent X-variable and a dependent Y-variable. L, H, and I describe the low end L, high end H, and interval between points in a set of values of the independent X-variable. Table-name is the name of an associated table, or set of constant values, of the dependent Y-variable that correspond to each of the values of the X-variable. Thus,

```
Y       = TABLE ( Table-1 , X , 0 , 5 , 1 )
Table-1 = 3 / 7 / 9 / 11/ 13/ 14
```

would represent the following functional relationship:

X	0	1	2	3	4	5
Y	3	7	9	11	13	14

Such table functions are used, as is shown below, to characterize the QA parameter profile of project EXAMPLE:

1. NOMINAL NUMBER OF ERRORS COMMITTED PER TASK (NERPK)

```
NERPK   = TABLE (Table-1, "% OF JOB WORKED", 0 , 100 , 20)
Table-1 = 25/23.86/21.59/15.9/13.6/12.5  ERRORS/KDSI
```

2. NOMINAL QA MANPOWER NEEDED PER ERROR (NQAMPE)

```
NQAMPE = TABLE (Table-2, "% OF JOB WORKED", 0 , 100 , 10)
Table-2 = .4/.4/.39/.375/.35/.3/.25/.225/.21/.2/.2
                                          Man-Days/ERROR
```

3. NOMINAL REWORK MANPOWER NEEDED PER ERROR (NRWMPE)

```
NRWMPE  = TABLE (Table-3, "% OF JOB WORKED", 0 , 100 , 20)
Table-3 = .6/.575/.5/.4/.325/.3    Man-Days/ERROR
```

Bibliography:

1. Abdel-Hamid, T.K. "The Dynamics of Software Development Project Management: An Integrative System Dynamics Perspective." Unpublished Ph.D. dissertation, Sloan School of Management, MIT, January, 1984.

2. Abdel-Hamid, T.K. and Madnick, S.E. Software Project Management. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., to be published in 1987.

3. Abdel-Hamid and Madnick, S.E. "An Integrative System Dynamics Perspective to Software Project Management: Arguments for an Alternative Research Paradigm." Submitted for publication to the MIS Quarterly, December, 1986.

4. Alberts, D.S. "The Economics of Software Quality Assurance." National Computer Conference, 1976.

5. Belford, P.C., et al. "An Evaluation of the Effectiveness of Software Engineering Techniques." IEEE COMPCON, Fall, 1977.

6. Boehm, B.W. Software Engineering Economics, Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1981.

7. Boehm, B.W., et al. "Some Experiences with Automated Aids to the Design of Large-Scale Reliable Software." Proceedings of the International Conference on Reliable Software, April, 1975.

8. Buckley, F. and Poston, R. "Software Quality Assurance." IEEE Trans. on Software Engineering, (January, 1984), 36-41.

9. Chow, T.S. (ed.) Software Quality Assurance: A Practical Approach. Silver Spring, MD: IEEE Computer Society Press, 1985.

10. Cooper, J.D. and Fisher, M.J., (eds.) Software Quality Management. New York: Petrocelli Book, Inc., 1979.

11. Cougar, J.D. and Zawacki, R.A. Motivating and Managing Computer Personnel. New york: John Wiley & Sons, 1980.

12. DeMarco, T. Controlling Software Projects. New York: Yourdon Press, Inc., 1982.

13. Endres, A.B. "An Analysis of Errors and their Causes in System Programs." IEEE Transactions on Software Engineering. June, 1975, 140-149.

14. Fagan, M.E. "Design and Code Inspections to Reduce Errors in Program Development." IBM Systems Journal, Vol. 15, No. 3, 1976.

15. Forrester, J.W. Industrial Dynamics. Cambridge, Mass: The MIT Press, 1961.

16. Hart, J.J. "The Effectiveness of Design and Code Walkthroughs." The Sixth International Computer Software and Applications Conference (COMPSAC), November, 1982.

17. Jones, T.C. "Measuring Programming Quality and Productivity." IBM Systems Journal, Vol. 17, No. 1, 1978, 39-63.

18. Knight, B.M. "Organizational Planning for Software Quality." In Software Quality Management. Edited by J.D. Cooper and M.J. Fisher. New York: Petrocelli Books, Inc., 1979.

19. Lawler, R.W. "System Perspective on Software Quality." In Software Quality Assurance: A Practical Approach. Edited by T.S. Chow. Silver Spring, MD: IEEE Computer Society Press, 1985.

20. McClure, C.L. Managing Software Development and Maintenance. New York: Van Nostrand Reinhold Company, 1981.

21. Mitchell, J.R. "Observations on the Use of Seven Structured Programming Techniques." IEEE, 1980.

22. Myers, G.J. Software Reliability: Principles and Practices. New York: John Wiley & Sons, Inc., 1976.

23. Nelson, E.C. "Software Reliability, Verification and Validation." Proceedings of the TRW Synposium on Reliable Cost Effective, Secure Software, Redondo Beach, CA: TRW, Inc., 1974.

24. Putnam, L.H. and Fitzsimmons, A. "Estimating Software Costs," Parts I,II, and III. Datamation, Sept., Oct., and Nov., 1979.

25. Radice, A. "Productivity Measures in Software." The Economics of Information Processing Volume (2): Operations, Programming and Software Models. Edited by R. Goldgerg and H. Lorin. New York: John Wiley & Sons, Inc., 1982.

26. Richardson, G.P. and Pugh, G.L. III. Introduction to System Dynamics Modeling with Dynamo. Cambridge, Mass: The MIT Press, 1981.

27. Riggs, H.E. Managing High-Technology Companies. Belmont, CA: Lifetime Learning Publications, 1983.

28. Roberts, E.B. (ed.). Managerial Applications of System Dynamics. Cambridge, Mass: The MIT Press, 1981.

29. Shneiderman, B. Software Psychology - Human Factors in Computer and Information Systems. Cambridge, MA: Winthrop Publishers, Inc., 1980.

30. Shooman, M.L. Software Engineering - Design, Reliability and Management. New York: McGraw-Hill, Inc., 1983.

31. Shooman, M.L. and Natarajan, S. "Effect of Manpower Development and Bug Generation on Software Error Models." Rome Air Development Center, RADC-TR-76-400, Jan., 1977.

32. Thayer, R.H. "Modeling a Software Engineering Project Managaement System." Unpublished Ph.D. dissertation, University of California, Santa Barbara, 1979.

33. Thibodeau, R. and Dodson, E.N. "Life Cycle Phase Interrelationships." Journal of Systems and Software, Vol. 1, 1980, 203-211.

34. Tsui, F. and Priven, L. "Implementation of Quality Control in Software Development." National Computer Conference, 1976.

35. Weinberg, G.M. The Psychology of Computer Programming. New York: Litton Educational Publishing, Inc., 1971.

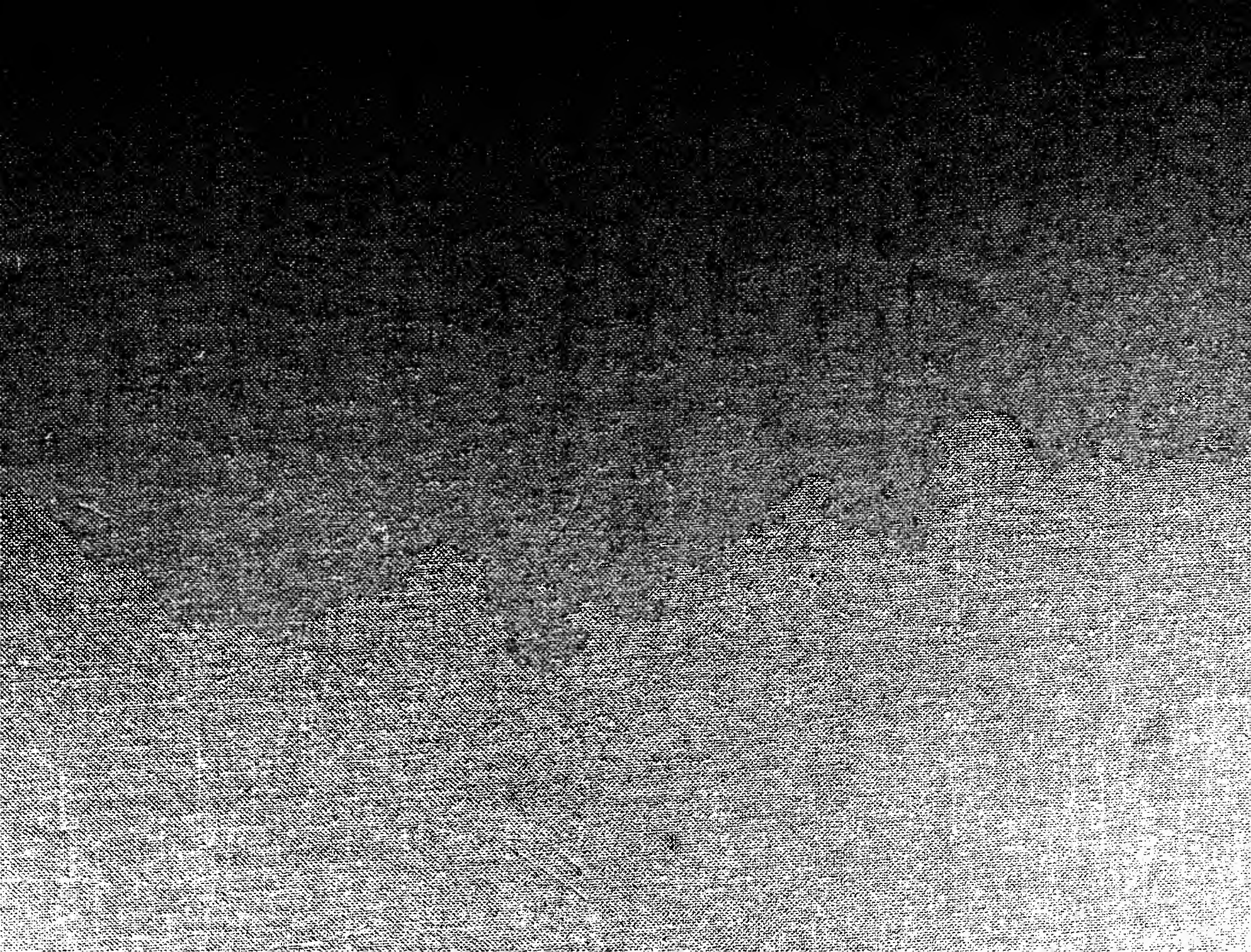
36. Weiss, D.M. "Evaluating Software Development by Error Analysis." Journal of Systems and Software, Vol. 1, 1979, 57-70.

37. Weiss, D.M. A Comparison of Errors in Different Software Development Environments. A Report for the Naval Research Lab, Washington, D.C., July, 1982.

4953 055

Date Due

MAR 2 1991

Lib-26-67

3 9080 004 231 160